‘Programação Funcional e em Lógica’ **2024/25**

-- Class 14 – Group 07 ---------- Contribution --

| | |
|---|---|
| Tiago de Pinho Bastos de Oliveira Pinheiro – UP202207890 | 50% |
| Tiago Grilo Ribeiro Rocha – UP202206232 | 50% |

---

- Other than the last couple functions, we tried to split the work as evenly as possible between the two, still helping each other out when needed even in a function that wasn’t assigned to us.

- With the last two, each of us spent a little more time than the other on a different one, but still helped each other when needed.

## Function ‘shortestPath’

The shortestPath function finds the shortest path—or paths, if there are multiple options with the same minimum distance—between two cities in a given network.

We chose a **BFS** algorithm because it’s effective for finding the shortest paths, and even though it’s usually used with unweighted graphs, it also works well with graphs weighted with distances only.

We also used elements like queues and sets because of their efficiency in saving information needed to prevent mistakes and/or unnecessary cycles.

## How it works

Firstly, we check if both the cities given are the same, and if so, we return that city alone as an output.

If both given inputs are different, we begin a **BFS search** by creating a queue with a single tuple, which we then place the **start** city inside of along with a distance of **0**. We create an empty set of visited cities, so we can track which cities we already stopped by in each iteration, as well as an empty list that will keep all the paths found with the minimum distance from the **start** city to the **end** city and a single variable that will hold the value of the minimum distance found.

Having finished those preparations, we begin the BFS search, by taking the first path from the queue. We saved the last city reached along with the distance it took to reach it. We check if that city is the **end** city given to us as an input, and if it is we need to check if it’s a possible shortest path. We compare the saved distance to the current minimum distance, and get four possible outcomes:

- Minimum distance is still 0, which means this is the first path found, and we save its distance as the current minimum distance, placing the path inside the shortest paths list.
- Saved distance is higher than the minimum distance stored, which means the path found is not the shortest discovered so far, so we discard it.
- Saved distance is equal to the minimum distance stored, which could mean there is more than one shortest path, so we simply add it to the shortest paths list and move on.
- Saved distance is lower than the minimum distance stores, meaning we found a new shortest path, so we replace the value of the minimum distance with the newfound distance, and reset the shortest path’s list so it only contains this new one.

If the current city we're checking isn't the **end** city, we continue exploring the possible paths from said city. We do this by expanding the search progressively to each of the current city's neighbors, using our *adjacent* function. We check if each neighbor hasn't been visited in the current path, and if not, we create a new path by adding it to the path, along with updating the distance to include it. We then add the path to the end of the queue.

We keep this up until the queue is empty and we have explored all possible paths. This means the list of shortest paths now contains the only possible shortest path(s) from the **start** city to the **end** city, which we return at the end.

## Auxiliary data structures used

Some auxiliary structures we used include, as previously mentioned, **queues**, which store a tuple with a path and its distance overall, used to help the BFS search, **sets**, used to keep track of previously visited cities so we avoid unnecessary cycles and keep the paths simple, **lists**, used to save the paths we know so far at each iteration that are possibly the shortest paths we want, in case there is more than one, and finally a **minimum distance tracker**, which we use to keep track of the shortest path distance found so far and compare it to any other possible paths we end up finding.

## Function ‘travelSales’

The travelSales function finds a solution to the traveling salesman problem by calculating the shortest path that visits each city exactly once and returns to the starting city. The point of the code is to find the minimum distance to be traveled across the entire path.

We use a **dynamic programming approach** with memoization to efficiently calculate the path while avoiding unnecessary or redundant results. By saving previously calculated subpaths, we can handle intermediate results more quickly.

We also use some helper functions, such as **findTSP** to make recursive exploration and **buildAdjList** to help manage city connections.

## How it works

Firstly, we call the buildAdjList function to turn the RoadMap into an adjacency list, which is then stored as a map where each city has its own adjacent cities and their distances associated with it.

We make the first city in the list of all cities be set as the start city and we initialize a dpTable, which is an empty map made to memoize the shortest paths found.

The main recursive function, findTSP, uses a TSP search with memoization, which explores all possible paths branching out from the current city, visiting only unvisited cities recursively. When a city is visited, it is added to the visited set, and only when all cities are visited does it search for a direct path back to the start city. It ends up calculating the minimum distance that ends up returning to the starting city.

The process mentioned above can be simplified into two different scenarios:

- Not all cities have been visited, which the function then proceeds to check the adjacent unvisited cities, recursively calling the function again on each new city while calculating its distance, to further extend the path being analyzed.

- All cities have been visited, the function tries to find a direct path to the start city, and if there is one, it returns this final distance while completing the cycle. If it can’t find a direct path, it returns nothing, which means there is no solution.

All these results are stored in a memoization table to avoid unnecessary calculations in future calls.

We then reverse the path list while adding the start city at the end to complete the cycle, returning the shortest possible path. If it doesn't find any valid path, it returns an empty list.

## Auxiliary data structures used

Some auxiliary structures we used include **adjacency lists**, which store each city's neighbors and their distances, **sets**, used to assure each city is only present once for each path, **memoization tables**, used to save calculated shortest paths and avoid redundant new calculations.

We also used some helper functions such as **adjacentCities**, which gets the adjacent cities for any given city, **lookupDistance**, which helps find the direct distance between two cities and **minimumBy**, which is used to find the minimum distance path from various given paths.